Generics in Java

type-safe

- *Problema:* garantire a compile time la coerenza dei dati di una collezione

- Il problema, noto come "*type-safe*", richiede la capacità di usare strutture ed algoritmi generali specializzandoli al volo

Non è banale!

- Il controllo di tipo avviene sicuramente a run-time, ma potrebbe essere troppo tardi!

Quando la conformità può creare problemi…

- Mediante la conformità è possibile usare le sottoclassi come fossero classi base

- Ciò produce indubbi vantaggi, consentendo di utilizzare lo stesso algoritmo per tutte le classi derivate

- Si pensi ad esempio alle strutture dati Java (ad es. `Vector`), che sono progettate per accettare qualunque tipo di oggetto…

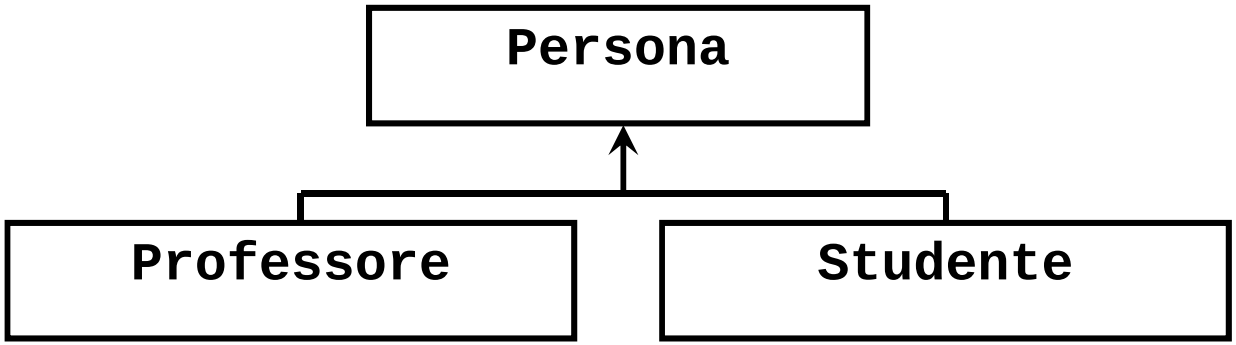
La conformità può creare problemi di type-safe

Un esempio concreto

- La soluzione si ha usando i *tipi parametrizzati* (chiamati in Java *generics*)
- Per meglio comprendere, si farà uso del seguente esempio:
  - si dispone di due tipi di persone: **studenti** e **professori**
  - si deve costruire un **archivio** capace di contenere sia studenti che professori
  - l'archivio deve essere **omogeneo** (non si devono mischiare professori con studenti e viceversa)

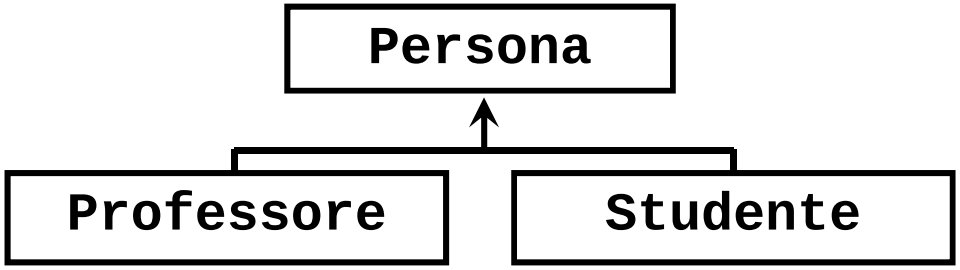
Un esempio concreto: implementazione della gerarchia

- Studenti e professori sono accomunati dal fatto di essere entrambi **persone**; inoltre si deve costruire una logica uguale per entrambe le specializzazioni
- Ciò porta alla realizzazione di un legame fra studenti e professori: una **gerarchia**!



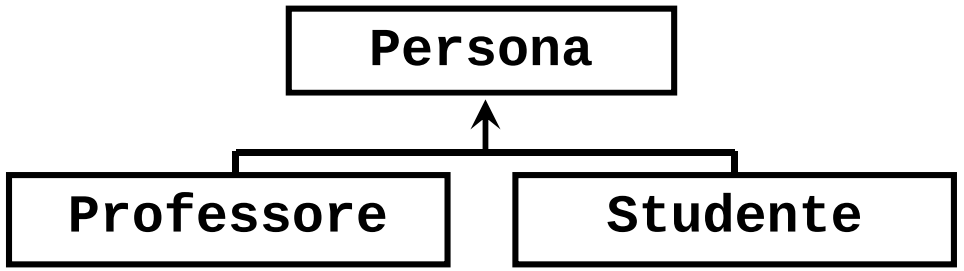
Un esempio concreto: implementazione della gerarchia

```
public class Persona {
    protected String nome, cognome;
    protected int eta;

    public Persona(String nome, String cognome,
                                int eta){
        this.nome = nome;
        this.cognome = cognome;
        this.eta = eta;
    }
```



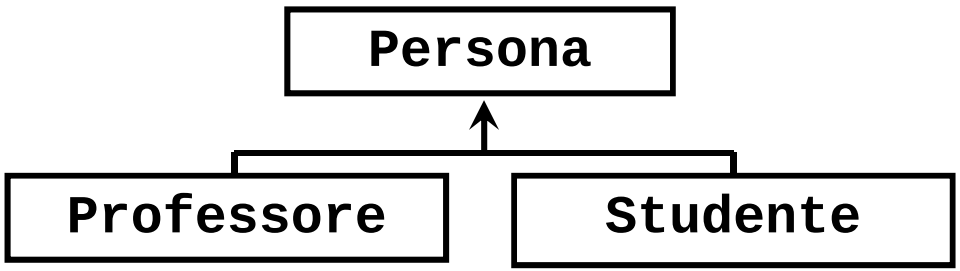
Un esempio concreto: implementazione della gerarchia

```
    public String toString(){
  return nome+" "+cognome+" "+eta;
    }
}  // fine della classe Persona
```



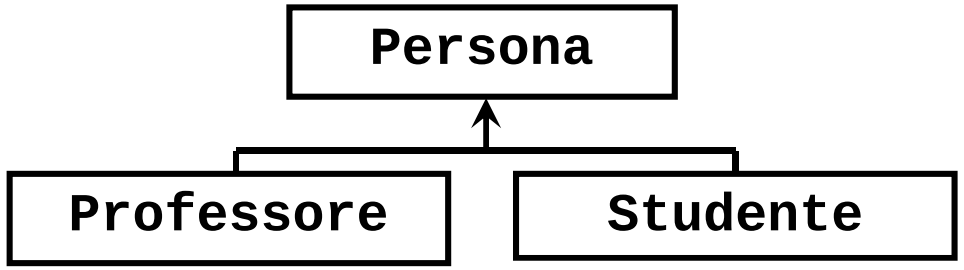
Un esempio concreto: implementazione della gerarchia

```
public class Professore extends Persona{
    public Professore(String nome,
                      String cognome, int eta) {
        super(nome,cognome,eta);
    }

    public String toString(){
      return nome+" "+cognome+" "+eta+
                     " - professore";
    }

}
```



Un esempio concreto: implementazione della gerarchia

```
public class Studente extends Persona{
    public Studente(String nome,
                    String cognome,
                       int eta) {
        super(nome,cognome,eta);
    }
}
```



Un esempio concreto: l'implementazione dell'archivio

```
import java.util.Vector;
public class Archivio1 {
    protected Vector persone;

    public Archivio1() {
        persone = new Vector(10);
    }

    public void aggiungi(Persona p){
        persone.add(p);
    }

    public void rimuovi(Persona p){
        persone.remove(p);
    }
```

Un esempio concreto: l'implementazione dell'archivio

```
public Persona get(int index){
        return (Persona)persone.get(index);
    }

    public int size(){
        return persone.size();
    }

} // fine della classe Archivio1
```

Il problema di `Archivio1`

- La classe `Archivio1` gestisce (con una sola logica) **tutti** i tipi `Persona`; ciò può provocare **type-unsafety**
- In altre parole non vi è nessun meccanismo che **controlli** il tipo (specifico) dei dati inseriti/rimossi dall'archivio

- Cosa ancora più importante: le incoerenze non possono essere rilevate a tempo di **compilazione**!

(si ricordi che `Studente` e `Professore` sono sottoclassi di `Persona`)

Un utilizzo scorretto

```
public static void main(String argv[]){
   // creazione di studenti e professori
   Studente s1 = new Studente("Luca","Ferrari",26);
   Studente s2 =
           new Studente("Santi","Caballe",29);
   Studente s3 =
           new Studente("James","Gosling",50);
   Professore pr1 =
           new Professore("Silvia","Rossi",27);
   Professore pr2 =
           new Professore("Simon","Ritter", 40);

   // creazione di due archivi separati
   Archivio1 archivio_prof = new Archivio1();
   Archivio1 archivio_stud = new Archivio1();
```

Un utilizzo scorretto

```
// aggiungo i prof e gli studenti ai relativi
  archivi
        archivio_prof.aggiungi(pr1);
        archivio_prof.aggiungi(pr2);
        archivio_prof.aggiungi(s1);

        archivio_stud.aggiungi(s1);
        archivio_stud.aggiungi(s2);
        archivio_stud.aggiungi(s3);
        archivio_stud.aggiungi(pr1);
```

Disastro imminente!

Disastro imminente!

Oggetti di tipo **incoerente** sono stati aggiunti agli archivi. Il compilatore e il sistema run-time **non** possono rilevare questo errore di logica, essendo l'archivio basato sulla **superclasse** dei tipi realmente utilizzati.

Un utilizzo scorretto

```
// stampa professori
for(int i=0;i<archivio_prof.size();i++){
  Professore pTemp =
  (Professore)archivio_prof.get(i);
  System.out.println("Professore "+i+" "+pTemp);
}

// stampa studenti
for(int i=0;i<archivio_stud.size();i++){
  Studente sTemp =
   (Studente)archivio_stud.get(i);
  System.out.println("Studente "+i+" "+sTemp);
      }
   }    // fine del metodo main
```

Un utilizzo scorretto

- Il problema risiede nel cast fatto al momento dell'estrazione dall'archivio:

```
Professore pTemp = (Professore)archivio_prof.get(i);
```

- L'assunzione è **corretta**: l'archivio dei professori (studenti) dovrebbe contenere solo professori (studenti), quindi il cast esplicito è lecito

output di esecuzione

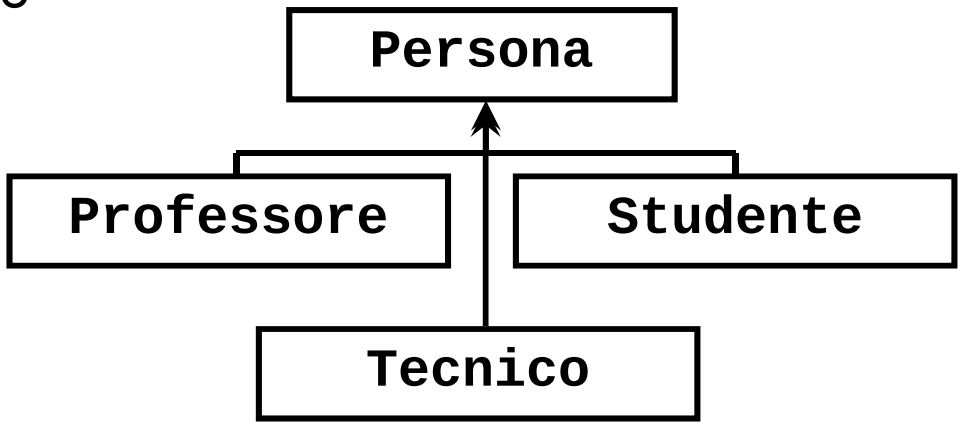
```
Professore 0 Silvia Rossi 27 – professore

Professore 1 Simon Ritter 40 - professore

Exception in thread "main“ java.lang.ClassCastException:

 seminario_20.generics.Studente

        at seminario_20.generics.Main1.main(Main1.java:32)
```

Possibili soluzioni

1. Realizzare una forma di archivio **specifica** per ogni tipo di dato presente

   Ad esempio, realizzando un archivio che accetti come dato un `Professore`, non si potrà correre il rischio di inserirvi uno `Studente`

Questa soluzione risulta valida nel **breve termine**, ma non accettabile nel lungo periodo, essendo incapace di gestire correttamente il code refactoring. Ogni volta che un nuovo tipo viene inserito, un nuovo archivio deve essere implementato



Possibili soluzioni

2. utilizzare **interfacce** differenti per ogni archivio

Definendo una **interfaccia** per ogni archivio, e "costringendo" l'archivio ad implementarle entrambe, si ottiene una soluzione molto Object Oriented, abbastanza **sicura** e sicuramente **flessibile**, al costo di una **minima aggiunta di codice**.

Possibili soluzioni

2. utilizzare interfacce differenti per ogni archivio

```
public interface archivio_studenti{
       public void aggiungi(Studente s);
       ...
}

public interface archivio_professori{
       public void aggiungi(Professore p);
}

public class Archivio
implements archivio_studenti, archivio_professori
{...}
```

Possibili soluzioni

2. utilizzare interfacce differenti per ogni archivio

I **problemi** di questa soluzione:

a) è potenzialmente **non type-safe**, essendo comunque possibile scavalcare le interfacce per lavorare direttamente sull'archivio
b) come per la soluzione precedente, ogni volta che un nuovo tipo viene **aggiunto**, l'archivio deve essere modificato di conseguenza

Possibili soluzioni

3. fornire un **parametro** all'archivio che imponga il tipo di istanza da gestire

```
public Archivio1(Class clazz){
       this.managedType = clazz;
}

public void aggiungi_persona(Persona p) throws Exception{
       if( managedType != null &&
           (p.getClass()!= managedType )  ){
             throw new Exception("Tipo sbagliato");
       }
       // inserimento dell'argomento…
}
```

Anticipa ciò che avviene a run-time
per un cast incorretto!

Possibili soluzioni

4. **eliminare** la gerarchia

Non è una soluzione Object Oriented!
Produce la scrittura di **molto codice** in più

Generics

- Dalla versione 5 di Java è possibile un controllo accurato sui tipi a **tempo di compilazione**
- ma le strutture dati devono essere progettate come *tipi parametrici* (***generics***)

- Una classe che sfrutti generics non ha un tipo di dato predefinito, ma lo **riceve** a tempo di **istanziazione**

Generics

- In altre parole, con generics, è possibile costruire classi con **algoritmi standard** e capaci di **agire su più tipi di dati**, ma in modo **coerente**

- "From the perspective of both software engineering and programming pedagogy, Java has a crude type system. Its most significant failing is the lack of support for generic types."

(Erica Allen, Safe Instantiation in Generic Java, PPPJ 2004)

- La sintassi di generics fa uso delle **parentesi angolari** (<,>) per indicare un **tipo** da specificare in seguito

Usando generics...

```
import java.util.Vector;
public class Archivio2<E> {
   protected Vector persone;

    public Archivio2() {
        persone = new Vector(10);
    }

    public void aggiungi(E p){
        persone.add(p);
    }

    public void rimuovi(E p){
        persone.remove(p);
    }
```

Con questa dichiarazione si indica che l'identificatore E fa riferimento ad un tipo di dato che sarà specificato nel codice che userà Archivio2

I metodi ora fanno riferimento a variabili di tipo E

Usando generics...

```
    public E get(int index){
        return (E)persone.get(index);
    }

    public int size(){
        return persone.size();
    }

} // fine della classe Archivio2
```

Usando generics…

```
public static void main(String argv[]){
   ...
    Archivio2<Professore> archivio_prof =
                      new Archivio2<Professore>();
    Archivio2<Studente> archivio_stud =
                      new Archivio2<Studente>();
    Archivio2<Persona>  archivio_per =
                      new Archivio2<Persona>();

    archivio_prof.aggiungi(pr1);
    archivio_prof.aggiungi(pr2);
    archivio_prof.aggiungi(s1);
```

output di compilazione

Main2.java:34: aggiungi(seminario_20.generics.Studente) in seminario_20.generics.Archivio2<seminario_20.generics.Studente> cannot be applied to (seminario_20.generics.Professore)

I vantaggi di generics

- Le **incoerenze** di tipo sono rilevate a tempo di **compilazione**, e non a tempo di esecuzione, aiutando il programmatore nel trovare errori di logica
- Si noti che generics non impedisce di usare la gerarchia mischiando i tipi, ma semplicemente richiede che si sia coscienti di ciò che si sta facendo:

```
Archivio2<Persona>  archivio_per =
                                  new Archivio2<Persona>();
```

Wildcards

- Generics ammette l'uso del carattere '?' come speciale wildcard, con i seguenti significati:
- `<? extends type>` indica tutti i tipi che ereditano da `type`.
  - Ad esempio `Archivio2<? extends Persona>` indica tutti i tipi di `Archivio2` parametrizzati da `Persona`
- `<? super type>` simile al caso precedente, ma tratta superclassi

Specializzare una classe con generics

- È possibile **ereditare** da una classe e aggiungere, nel contempo, il supporto a generics, facendo però attenzione affinché i metodi non siano in conflitto

```
public class Archivio3<E> extends Archivio1{
        public void aggiungiElement(E p){
          persone.add(p);
        }
    ...
}
```

ATTENZIONE: si ricordi che Studente (Professore) è anche Persona, quindi un metodo aggungi(E p) potrebbe andare in conflitto con Archivio1.aggiungi(Persona p) qualora il tipo sia ancora una Persona.

Cosa c'è dietro a generics

- La "magia" di generics risiede nel nuovo sistema di compilazione

- Il compilatore effettua ora alcuni passi di **manipolazione sintattica** (*type erasurers*) al fine di forzare eventuali errori di casting

Generics in azione: codice utilizzato

- Il tag di generics viene rimosso, e il codice viene compilato sostituendo al tipo parametrico **`Object`**

```
public class Archivio2<E> {
   protected Vector persone;

    public Archivio2() {
        persone = new Vector(10);
    }

    public void aggiungi(Object p){
        persone.add(p);
    }

    public void rimuovi(Object p){
        persone.remove(p);
    }
```

Generics in azione: codice utilizzatore

- Il compilatore rimuove i tag di generics, e "forza" dei cast

```
public static void main(String argv[]){
   ...

    Archivio2<Professore> archivio_prof =
                        new Archivio2<Professore>();
    Archivio2<Studente> archivio_stud =
                        new Archivio2<Studente>();
    Archivio2<Persona>  archivio_per =
                        new Archivio2<Persona>();

    archivio_prof.aggiungi((Professore)pr1);
    archivio_prof.aggiungi((Professore)pr2);
    archivio_prof.aggiungi((Professore)s1);
   ...
}
```

Generics e il resto del mondo

- La **libreria** Java supporta appieno generics:

```
public static void main(String argv[]){
        Vector<String> vettore = new Vector<String>();
        for(int i=0;i<10;i++){
            vettore.add(new String("stringa n."+i));
        }
        for(int i=0;i<10;i++){
            String s = vettore.elementAt(i);
            System.out.println(s);
        }
    }
```

Non devo fare il cast

Ma non tutto è generics!

- Se si tenta di utilizzare una classe "**normale**" come fosse generics, si ottiene un **errore di compilazione**

```
Archivio1<Professore> = new Archivio1<Professore>();
```

output di compilazione

```
Main4.java:15: not a statement
  Archivio1<Professore> = new Archivio1<Professore>();
Main4.java:15: ';' expected
  Archivio1<Professore> = new Archivio1<Professore>();
```

Il rovescio della medaglia

- Generics consente di usare un **algoritmo generico** in ***type-safe***, ma per impostazione predefinita **non** impedisce di usare l'algoritmo per istanze diverse da quelle per cui questo è stato progettato (cosa impedita dalla conformità)!

```
Archivio2<String> a = new
  Archivio2<String>();
```

- La ragione di ciò risiede nel modo in cui la classe sottoposta a generics viene compilata: tutti gli identificatori sono sostituiti con **`Object`**.

È possibile limitare i tipi utilizzabili!

Limitare l'uso dei tipi

```
public class Archivio2<E extends Persona>{…}
```

Se a questo punto si tenta di creare un archivio con un tipo sbagliato, si ottiene un errore di compilazione

```
Archivio2<String> archivio_stud =
                        new Archivio2<String>();
```

output di compilazione

```
seminario_20\generics\Main9.java:13: type parameter java.lang.String is not within its bound
        Archivio2<String> archivio_stud =
                                new Archivio2<String>();
```

Generics non implica relazioni!

- Tutte le istanze create in modo parametrizzato condividono la **stessa classe**

```
Archivio2<Persona> aPersona = …
Archivio2<Studente> aStudente = …
```

Non sono in relazione!

(anche se `Studente` eredita da `Persona`)

Generics extends Generics

- È possibile **estendere** una classe che fa uso di generics, la sottoclasse può a sua volta fare uso di generics

- Valgono tutte le **regole** dell'ereditarietà (es. overriding)!

- Esempio: estendere l'archivio visto in precedenza (`Archivio2`) in modo che possa memorizzare **associazioni** `Studente-Professore` in type-safe

Esempio

```
import java.util.Hashtable;

public class Archivio5<E,R> extends Archivio2<E>{
    // relazioni
    protected Hashtable relazioni;

    public Archivio5(){
  super();
  relazioni = new Hashtable();
    }

    public void aggiungiRelazione(E p1, R p2){
  relazioni.put(p1,p2);
    }
```

Esempio

```
public void stampaRelazioni(){
  Enumeration<E> chiavi = relazioni.keys();

  while(chiavi.hasMoreElements()){
      E chiave = chiavi.nextElement();
      System.out.println("Relazione "+chiave+" - "
  +relazioni.get(chiave));
  }
    }

}
```

Esempio: utilizzo

```
public class Main5{
    public static void main(String argv[]){
        Studente s1 =
            new Studente("Luca","Ferrari",26);
        Studente s2 =
            new Studente("Santi","Caballe",29);
        Studente s3 =
            new Studente("James","Gosling",50);

        Professore pr1 =
            new Professore("Silvia","Rossi",37);
        Professore pr2 =
            new Professore("S.","Ritter", 40);
```

Esempio: utilizzo

```
  Archivio5<Studente,Professore> archivio =
                new Archivio5<Studente,Professore>();

  archivio.aggiungi(s1);
  archivio.aggiungi(s2);
  archivio.aggiungi(s3);

  archivio.aggiungiRelazione(s1,pr1);
  archivio.aggiungiRelazione(s2,pr2);

  archivio.stampaRelazioni();
  }
}  // fine della classe
```

output di esecuzione

```
Relazione Luca Ferrari 26 - Silvia Rossi 37 - professore

Relazione Santi Caballe 29 – S. Ritter 40 - professore
```

Considerazioni sull'esempio

- È possibile utilizzare generics con più di un tipo di parametro:

```
public class Archivio5<E,R> extends Archivio2<E>{...}
```

```
Archivio5<Studente,Professore> archivio = new
  Archivio5<Studente,Professore>();
```

- Il type-safe è garantito:

```
archivio.aggiungi(pr1);
archivio.aggiungiRelazione(pr1,s1);
```

Risultato

output di compilazione

```
seminario_20/generics/Main5.java:17: aggiungi(seminario_20.generics.Studente) in
seminario_20.generics.Archivio2<seminario_20.generics.Studente> cannot be applied to
(seminario_20.generics.Professore)
        archivio.aggiungi(pr1);
                 ^
seminario_20/generics/Main5.java:20:
aggiungiRelazione(seminario_20.generics.Studente,seminario_20.generics.Professore) in
seminario_20.generics.Archivio5<seminario_20.generics.Studente,seminario_20.generics.Professore> cannot be applied
to (seminario_20.generics.Professore,seminario_20.generics.Studente)
        archivio.aggiungiRelazione(pr1,s1);
```

Templates? No grazie!

- Anche se molto simili nella sintassi e nell'utilizzo, i Java generics **non** sono la stessa cosa dei template C++
- I template C++ si riconducono a **macro del preprocessore**, che producono il codice sorgente di una nuova classe con i tipi "fissati"
- Generics opera a livello di **compilatore** e non "sporca" il codice della classe che si sta utilizzando